Discrete Applied Mathematics 157 (2009) 3485–3493

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

Polyhedral approximation and practical convex hull algorithm for certain classes of voxel sets☆

Henrik Schulz *

Forschungszentrum Dresden – Rossendorf, Department of Information Technology, PF 510119, 01314 Dresden, Germany

ARTICLE INFO

Article history:
Received 25 August 2008
Received in revised form 18 March 2009
Accepted 8 April 2009
Available online 6 May 2009

Keywords:
Digital geometry
Convex hull
Abstract cell complex
Abstract polyhedron
Surface approximation

ABSTRACT

In this paper we introduce an algorithm for the creation of polyhedral approximations for certain kinds of digital objects in a three-dimensional space. The objects are sets of voxels represented as strongly connected subsets of an abstract cell complex. The proposed algorithm generates the convex hull of a given object and modifies the hull afterwards by recursive repetitions of generating convex hulls of subsets of the given voxel set or subsets of the background voxels. The result of this method is a polyhedron which separates object voxels from background voxels. The objects processed by this algorithm and also the background voxel components inside the convex hull of the objects are restricted to have genus 0. The second aim of this paper is to present some practical improvements to the discussed convex hull algorithm to reduce computation time.

1. Introduction

A problem often arising in the field of three-dimensional image analysis is the efficient encoding of the surface of a digital object which is given as a set of voxels. The most popular approach is that of triangulation, not only because of the simplicity of triangles but also because of the existing hardware support for tasks in the field of computer graphics.

A widely used approach to triangulate voxel objects is the Marching Cubes Algorithm by Lorensen and Cline [9]. It has a very low time complexity, i.e. it is linear in the number of voxels, which makes this algorithm applicable in practical tasks. But it has also two important drawbacks. First, the number of generated triangles is in most cases greater than the number of surface elements (faces) of the original voxel image and second, the orientation of the triangles is limited to a few directions. This is not desirable when an approximation of the original object (before digitization) is needed, which has a smooth surface with a constant curvature, for example.

Other triangulation methods use a divide-and-conquer approach. The algorithm described in [3] is applicable not only in two-dimensional spaces to produce Delaunay triangulations, but also in higher dimensions, i.e. also in the three-dimensional case. The algorithm separates the input data into two subsets and constructs the triangulation of the subsets recursively.

Sometimes there exists the necessity to generate a more economical surface than a triangulation. Especially when triangulations would have a lot of coplanar triangles, a polyhedral surface (a surface containing faces with more than three edges) would be much more efficient. The problem of approximating polyhedral surfaces is also well studied in the field of computational geometry [1,2,4].

☆ A preliminary version of this paper was presented at the 12th International Workshop on Combinatorial Image Analysis, Buffalo, NY, USA, April 2008 [H. Schulz, Polyhedral surface approximation of non-convex voxel sets through the modification of convex hulls, in: V.E. Brimkov, R.P. Barneva, H.A. Hauptman (Eds.), Combinatorial Image Analysis, in: Lecture Notes in Computer Science, vol. 4958, Springer-Verlag, 2008, pp. 38–50].

* Tel.: +49 3512603268; fax: +49 35126013268.
E-mail address: h.schulz@fzd.de.

0166-218X/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.dam.2009.04.008

In [8] we have already shown that for a convex voxel set the convex hull is such a polyhedral surface. In this paper we present an improvement to this algorithm to handle not only convex voxel sets, but also voxel sets with cavities and concavities (definitions are given below). The voxel sets and also the subsets of the background voxels inside the convex hull are restricted to have genus 0. Furthermore the objects must be strongly connected and homogeneously three-dimensional subsets of an abstract cell complex (AC complex).

The formal task to be solved is the following: Given a set V of voxels. V shall be a strongly connected three-dimensional subcomplex of an AC complex, i.e. any two voxels of V may be connected by a path of pairwise incident cells of the form $c_0^3 c_1^2 c_2^3 \dots c_{l-1}^2 c_l^3$, where c_i^3 is a three-dimensional cell of V and c_j^2 is a two-dimensional cell of V. We want to create a closed polyhedral surface H containing V with the minimum surface area. Since there can be more than one polyhedron with the minimum surface area, we search the one with the minimum number of faces.[1] The polyhedral surface H shall separate object voxels (interior) from background voxels (exterior) in such a way that no object voxel lies outside H and no background voxel lies inside H. Voxels lying on H, especially the vertices of the polyhedron, have to be marked as being object voxels or background voxels. For the first criterion (minimum surface area) we will only present an approximation here. The second criterion (separation) is stated to provide the possibility of exactly restoring the original voxel set V from the polyhedron H. An efficient data structure to store the polyhedral surface is the cell list [5].

In the next section we present the basic definitions used in this paper. In Section 3 we shortly present the algorithm for the construction of the convex hull and we give some new improvements to this algorithm concerning the reduction of the computation time. The main part of this paper follows in Section 4 where we present our method to construct polyhedral surfaces for non-convex digital objects through recursive repetitions of generating convex hulls of subsets of the given voxel set or subsets of the background voxels. In Section 5 we investigate the complexity of the presented algorithms. The paper ends with a conclusion in Section 6 and a bibliography.

2. Basic definitions

The algorithm presented here is based on the theory of AC complexes introduced into the field of image analysis by Kovalevsky [5]. Most of the basic notions of this theory relevant to the topic of polyhedral surfaces are gathered in the Appendix to [8].

Let V be a given set of voxels in a Cartesian three-dimensional space. The voxels of V are specified by their coordinates. Our aim is to construct the convex hull K of V and a modification H of K which represents the polyhedral surface of V. We consider the convex hull and the modified hull as abstract polyhedra according to the following definition:

Definition AP. An *abstract polyhedron* is a three-dimensional AC complex containing a single three-dimensional cell whose boundary is a two-dimensional combinatorial manifold without boundary. The two-dimensional cells (2-cells) of the polyhedron are its *faces*, the one-dimensional cells (1-cells) are its *edges* and the zero-dimensional cells (0-cells) are its *vertices* or points [8].

An abstract polyhedron is called a *geometric* one if coordinates are assigned to each of its vertices. We shall call an abstract geometric polyhedron an AG-polyhedron. Each face of an AG-polyhedron PG must be planar. This means that the coordinates of all 0-cells belonging to the boundary of a face F_i of PG must satisfy a linear equation $H_i(x, y, z) = 0$. If these coordinates are coordinates of some cells of a Cartesian AC complex A then we say that the polyhedron PG is embedded into A or that A contains the polyhedron PG.

Definition CP. An AG-polyhedron PG is said to be *convex* if the coordinates of each vertex of PG satisfy all the linear inequalities $H_i(x, y, z) \leq 0$ corresponding to all faces F_i of PG. The coefficients of the linear form $H_i(x, y, z)$ are the components of the outer normal of F_i [8].

A cell c of the complex A containing the convex AG-polyhedron PG is said to *lie in* PG if the coordinates of c satisfy all the linear inequalities $H_i(x, y, z) \leq 0$ of all faces F_i of PG.

Definition CH. The *convex hull* of a finite set V of voxels is the smallest convex AG-polyhedron PG containing all voxels of the set V. "Smallest" means that there exists no convex AG-polyhedron different from PG which contains all voxels of V and whose all vertices are in PG [8].

For the differentiation between voxel sets being convex or not, we need to define what a convex voxel set actually is.

Definition DCS. A *digital half-space* is the set of all voxels whose coordinates satisfy a linear inequality. A *digital convex subset* of the space is a non-empty intersection of digital half-spaces [8].

This definition implies that coordinates are assigned to the voxels, which is fulfilled since the set V is given as a subset of a Cartesian space.

[1] Example: The faces of a cube (squares) can be subdivided into coplanar triangles. The surface area does not change, but the number of faces increases.

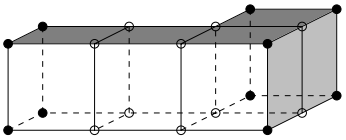

Fig. 1. Four voxels and their convex 0-cells (depicted as black disks). Non-convex 0-cells are depicted as circles. The voxel in the center of the front row is not incident to any convex 0-cell and thus it is not a candidate vector.

3. Convex hull

It is well known that every non-convex set can be considered as the union of convex sets. We use this property of non-convex sets to extend our incremental convex hull computing method presented in [8] to construct an abstract polyhedron from a non-convex set of voxels by subtracting small convex hulls from an initial convex hull. This is motivated by the imagination of modelling the surface through pressing faces of the convex hull onto the voxel object.

3.1. Constructing the convex hull

The first step to build a non-convex abstract polyhedron consists in creating the convex hull of the given voxel object V as described in [8]. The algorithm for constructing the convex hull consists of two parts: in the first part a subset of vectors v pointing to voxels must be found which are candidates for the vertices of the convex hull. The coordinates of the candidates are saved in an array L. The second part constructs the convex hull of the set L.

From the point of view of AC complexes the given set V is the set of three-dimensional cells (3-cells) of a subcomplex M of a three-dimensional Cartesian AC complex A. The complex A represents the topological space in which our procedure is acting. It is reasonable to accept that M is homogeneously three-dimensional according to the following definition:

Definition HN. An n-dimensional AC complex A is said to be *homogeneously n-dimensional* if every k-dimensional cell of A with $k < n$ is incident to at least one n-cell of A [6].

The problem of finding the vectors v can be defined as follows: A 0-cell is called a *convex* 0-cell iff it is incident to exactly one 3-cell of M (Fig. 1). All 3-cells incident to at least one convex 0-cell are the candidate vectors v. The vectors v are stored in L.

As already mentioned, the second part of our algorithm is that of constructing the convex hull of the set L of the candidate vectors v found by the first part.

To build the convex hull of L we first create a simple convex polyhedron spanning four arbitrary non-coplanar voxels v of L. It is a tetrahedron. It will be extended step by step until it becomes the convex hull of L. We call it the *current polyhedron*.

The next step in constructing the convex hull is to extend the current polyhedron while adding more and more voxels, some of which become vertices of the convex hull. When the list L of the candidate vectors is exhausted, the current polyhedron becomes the convex hull of M. The extension procedure is based on the notion of visibility of faces which is defined as follows.

Definition VI. The face F of a convex polyhedron is *visible* from a cell c, if c lies in the outer open half-space bounded by the face F, i.e. if the scalar product (N, w) of the outer normal N of the face F and the vector w pointing from a point Q in F to c, is positive. If the scalar product is negative then F is said to be *invisible* from c. If the scalar product is equal to zero then F is said to be *coplanar* with c [8].

To extend the current polyhedron the algorithm processes one voxel after another. For any voxel v it computes the visibility of the faces of the polyhedron from v. Consider first the simpler case when there are no faces of the current polyhedron, which are coplanar with v. The algorithm labels each face of the current polyhedron as being visible from v or not. If the set of visible faces is empty, then the voxel v is located inside the polyhedron and may be discarded. If one or more faces are visible, then the polyhedron is extended by the voxel v and some new faces. Each new face connects v with one edge of the boundary of the set of visible faces. A new face is a triangle having v as its vertex and one of the edges of the said boundary as its base. All triangles are included into the cell list of the current polyhedron while all visible faces are removed. Also each edge incident to two visible faces and each vertex incident only to visible faces is removed.

In Fig. 2 the boundary of the visible subset is shown by bold lines (solid or dashed). The edges shown by dotted lines must be removed together with the three faces visible from v. The remaining faces, edges and vertices are kept unchanged. The algorithm repeats this procedure for all voxels in L.

Consider now the problem of coplanar faces. There are three variants to treat a face coplanar with a new voxel v. The first variant treats coplanar faces as visible ones. In such a situation the algorithm removes all coplanar faces and replaces them by new triangles connecting the boundary edges with the voxel v. As a result the number of coplanar triangles is rather small.

Secondly, we can treat coplanar faces as invisible ones. In this situation the algorithm keeps the coplanar faces and creates more and more triangles. For a large number of coplanar voxels the number of faces created by this method is large, too.

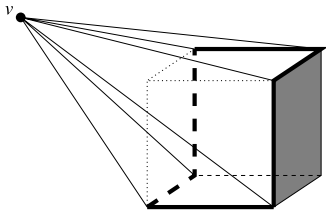

Fig. 2. The current polyhedron (a cube) being extended by the voxel v as a new vertex.

Both alternatives, treating coplanar faces as visible and invisible respectively, require a subsequent step of merging coplanar triangles since our aim is to produce a polyhedral approximation of the object surface, but not necessarily a triangulation. If a triangulation is desired, the step of merging coplanar faces must be omitted.

The third option to handle coplanar faces is a little bit more sophisticated and treats them neither as visible nor as invisible ones. We call it the *extension method* since the algorithm has to extend a coplanar face towards the new voxel v. This procedure is similar to the construction of the two-dimensional convex hull of v and the coplanar face. The result of this method of treating a coplanar face is a convex face with more than three edges. Then the number of generated faces is as small as possible.

If a triangulation is desired the best way is to treat coplanar faces as visible ones since the number of new faces is smaller than the number of faces created with the method which interprets coplanar faces as invisible ones.

The procedure of adding new faces to the current polyhedron ends after processing all candidate vectors in L. With this step the convex hull is completely constructed and the first part of creating a polyhedral surface of the given voxel object V ends.

3.2. Reducing computation time through sorting vertex candidates

The runtime of the convex hull algorithm depends considerably on the method of treating coplanar faces. But the computation time spent for the extension of coplanar faces is much less than the time saved by reducing the number of faces of the polyhedron.

Another runtime reduction can be achieved by a suitable selection of the vertices of the initial convex hull, i.e. the tetrahedron. The idea is to maximize the initial polyhedron and also the subsequent hulls to have many vertex candidates, which are not vertices of the convex hull, already located inside the polyhedron. For those vertex candidates the algorithm has to compute the visibility only. Since they are located inside the current polyhedron, there is no need to extend it, which means that no new faces have to be created. It is easy to see that this method is promising only for objects whose convex hulls have few vertices related to the number of vertex candidates.

For the maximization of the initial hull we first need to have the list L of vertex candidates sorted lexicographically. This is always the case since L is created by systematically going through the cell complex to find the candidates. Hence the first vector $\vec{v}_1$ in L is the one with minimum x-coordinate and minimum y-coordinate amongst all vectors with this x-value and minimum z-coordinate amongst all with this y-value. We cannot take the absolute minimum values for all three coordinates since such a cell does not necessarily belong to the object. The first candidate in L is also the first one in our new sorted list L_s. The second candidate $\vec{v}_2$ is the vector with maximum x-coordinate and minimum y- and z-coordinates chosen in the same way as for the first candidate. If there is one vector with minimum x-, y- or z-coordinate only then the next greater value has to be taken. For the third and the fourth candidate $\vec{v}_3$ and $\vec{v}_4$ we choose the ones with maximum y-coordinate and z-coordinate respectively. That is, the four vertices of the tetrahedron are:

$$\vec{v}_1 = \begin{pmatrix} x_{\min} \\ y_{\min} \\ z_{\min} \end{pmatrix}, \quad \vec{v}_2 = \begin{pmatrix} x_{\max} \\ y_{\min} \\ z_{\min} \end{pmatrix}, \quad \vec{v}_3 = \begin{pmatrix} x_{\min} \\ y_{\max} \\ z_{\min} \end{pmatrix}, \quad \vec{v}_4 = \begin{pmatrix} x_{\min} \\ y_{\min} \\ z_{\max} \end{pmatrix}.$$

The next four vertex candidates in L_s are chosen in a similar way with maximum values in their coordinates, i.e.:

$$\vec{v}_5 = \begin{pmatrix} x_{\max} \\ y_{\max} \\ z_{\max} \end{pmatrix}, \quad \vec{v}_6 = \begin{pmatrix} x_{\min} \\ y_{\max} \\ z_{\max} \end{pmatrix}, \quad \vec{v}_7 = \begin{pmatrix} x_{\max} \\ y_{\min} \\ z_{\max} \end{pmatrix}, \quad \vec{v}_8 = \begin{pmatrix} x_{\max} \\ y_{\max} \\ z_{\min} \end{pmatrix}.$$

With these eight vectors the algorithm produces an octahedron which is extended with all other candidate vectors of L_s containing the remainder of all cells of L in the same order.

As already mentioned this method has no benefit for objects whose all candidates are vertices of the convex hull. Digital balls are examples for such objects. But the computation time for choosing eight elements from a sorted list is negligibly small in relation to the advantage of saving a lot of work spent for creating and removing faces. In the general case, especially when considering non-convex objects, the number of vertices of the convex hull is significantly smaller than the number of vertex candidates.

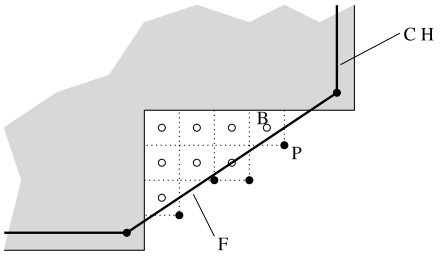

Fig. 3. Convex hull *CH* of an object (shaded in gray). Some of the 0-cells of the background voxel component are located outside *CH* and thus there exists a visible face.

4. Non-convex sets of voxels

4.1. Finding concavities

As already mentioned, the convex hull is a good means to encode the surface of a convex object. The convex hull of a digital convex set of voxels (see Definition DCS) never contains voxels of the background. If the given voxel object is not convex, we have to find components of the set of background voxels included into the convex hull. These components can be cavities, concavities and tunnels [13].

Definition CO. A *concavity* is a component of background voxels inside the convex hull which intersects exactly one connected set of faces of the convex hull.

Due to this definition a concavity does not have to be a convex protrusion of the background, i.e. it does not have to be convex.

Definition CA. A *cavity* is a component of background voxels inside the convex hull, which does not intersect any face of the convex hull.

Definition TU. A *tunnel* is a component of background voxels inside the convex hull which intersects more than one connected set of faces of the convex hull. If the tunnel intersects exactly two connected sets it is called a *non-branched* tunnel. If the number of connected sets of faces of the convex hull which are intersected by the component is greater than two, the tunnel is called a *branched* one.

Here we only deal with concavities and cavities. The latter is a trivial problem and will be mentioned later on. Tunnels are part of future work.

We use the algorithm described in [7] to find the components of the set of background voxels inside the convex hull and classify them by the number of connected sets of faces of the convex hull which are intersected by the component. To check whether a component of background voxels intersects a set of faces of the convex hull, we just have to compute the visibility of faces from the 0-cells of the background component. If every 0-cell of a component has no visible faces then the component is entirely located inside the hull and thus it is a cavity. If one or more 0-cells have visible faces or one or more 0-cells are coplanar with a set of faces then the component intersects the hull, i.e. it is a concavity or a tunnel (see Fig. 3).

The components are labeled and thus we can modify the convex hull by treating one component of background voxels after another.

4.2. Modification of the convex hull

After all components of background voxels inside the convex hull are found, we can modify the convex hull. As already mentioned, the convex hull is a convex polyhedron. After the first modification we can no longer speak of the convex hull, because it is no longer a convex polyhedron. Hence we call the modified hull *current polyhedron* again.

To modify the current polyhedron we first need to know which faces are intersected by the current background voxel component. This can be determined by using again the notion of visibility. In the previous step we have labeled a component if its 0-cells have visible faces and now we label the faces which are visible from the 0-cells of the current background voxel component. This means that a face F becomes labeled if the following criteria are all satisfied:

(1) The face F is visible from some of the 0-cells of the background voxel component or some 0-cells are coplanar with the face F.
(2) If there is a 0-cell P outside H and a background voxel B inside or on H with $P \in Cl(B)$, then the projection of B onto the face F is inside the boundary of F.
(3) The 0-cell P does not lie on the boundary of the face F.

We want to mention that the steps of labeling the background voxel components and labeling the corresponding faces can be merged together.

A topology preserving operation called "pressing-in" is applied to the set of labeled faces (see Fig. 4).

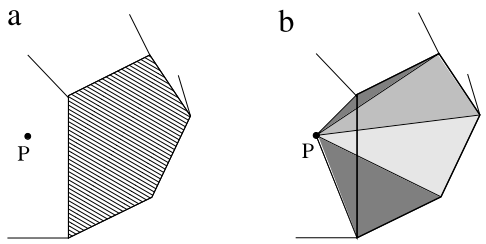

Fig. 4. A cell P inside the polyhedron (a) and the resulting polyhedron after pressing-in (b).

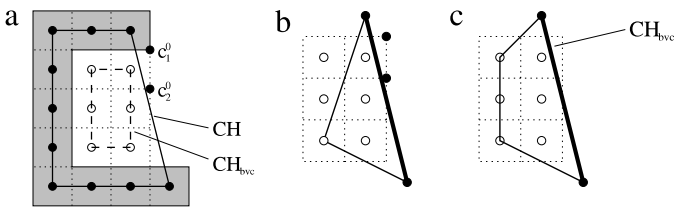

Fig. 5. (a) Convex hull CH of a given set of voxels and the convex hull CH_{bvc} of the background voxel component. Two 0-cells c_1^0 and c_2^0 of the background voxel component are located outside CH. (b) Initial convex hull of the set of background voxels created from a fixed face (bold line). (c) Resulting convex hull CH_{bvc} of the background voxels.

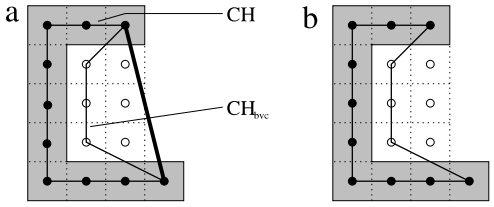

Fig. 6. CH and CH_{bvc} with a common face (a) and the resulting polyhedron after the modification step (b).

Definition PR. *Pressing-in* towards a non-empty set of cells located inside a polyhedron H is a topology preserving operation which replaces a connected set S_1 of faces of H by a new connected set S_2 of faces in such a way that the boundaries of the sets S_1 and S_2 are identical [11].

In the simplest case the set S_1 consists of one face only and thus it can be interpreted as the base of a pyramid which has the set S_2 as its sides. The apex of the pyramid (P in Fig. 4) is located inside the polyhedron. In the general case the set S_1 consists of several faces and the destination of the pressing-in is not necessarily a single cell.

We perform the pressing-in by constructing a polyhedron around the background voxel component. As mentioned in the Introduction, we want to apply our convex hull algorithm recursively to modify the convex hull of the voxel object. This means that we now create the convex hull of the background voxel component and modify it again and again until there are no object voxels outside the polyhedron and no background voxels inside it. To do so we have to combine the cell lists of the current polyhedron and that of the current polyhedron of the background voxel component. But this is not trivial. Definition PR implies that we can identify the faces of both polyhedra, but this is not possible in the general case (see Fig. 5a).

To avoid identifying faces of these two polyhedra, which do not have identical boundaries, we do not construct the convex hull of the background voxel component independently. A more precise approach consists in spanning the convex hull by starting with the labeled faces of the current polyhedron. This means that we span an initial polyhedron with this set of faces and a voxel of the background component, which lies inside the set of labeled faces (Fig. 5b). This initial polyhedron can be extended in the same way as the tetrahedron being the initial convex hull. The result is the convex hull of the set of labeled faces and the set of voxels of the background voxel component lying inside the polyhedron before applying the pressing-in operation (Fig. 5c).

At this stage of the modification we have the current polyhedron and a second smaller polyhedron which has a connected set of faces in common with the first one (see Fig. 6a). Now we apply our idea of recursivity by interpreting the second polyhedron as the current polyhedron and we change the roles of object and background voxels. Now we can apply the same algorithm to the new current polyhedron and thus we search for object voxels inside this polyhedron for which we have to do a pressing-in. This leads to a protuberance to the outside of the first polyhedron.

The result of the algorithm is a polyhedron $\hat{H}$ which separates object voxels from background voxels meaning that no object voxel lies outside $\hat{H}$ and no background voxel lies inside $\hat{H}$ (see Fig. 6b). Voxels lying on $\hat{H}$, especially the vertices of

$\hat{H}$, have to be marked in the cell list as being object or background voxels to preserve the possibility of reconstructing the object from the cell list of $\hat{H}$.

As already mentioned in Section 4.1, the algorithm can also deal with cavities, because it is a trivial problem. According to Definition CA cavities have no connection to the outer surface of the polyhedron and thus we can independently compute the convex hull of this background voxel component and modify it, if the cavity is a non-convex one. After applying the algorithm to the cavity we have to change the orientation of the normal vectors of the faces to ensure that they point to the outside of the surface of the voxel object, which means that they point to the inside of the cavity.

Given an object represented as subcomplex S of an AC complex A the modification algorithm can be summarized as follows:

```
create H=CH(S)
if H separates S and A\S {
  return H
} else {
  modify_polyhedron(H)
  return H
}
function modify_polyhedron(H) {
  find concavities inside H
  for each concavity S' {
    create CH(S')
    if CH(S') separates S' and A\S' {
      merge H and CH(S')
    } else {
      change roles of object and background
voxels
      modify_polyhedron(CH(S'))
    }
  }
  return H
}
```

Algorithm 1. The modification algorithm.

It is easily seen from the algorithm pseudocode that the complexity of this method depends not only on the number of voxels but also on the number of concavities and their complexity. A complete study on the complexity of the algorithm will be given in Section 5.

We want to mention that our algorithm has an important drawback. At this level of development it is not able to deal with a class of objects whose surfaces have a genus greater than 0 or whose background voxel components have a surface with a genus greater than 0, such as tori or mushrooms. This is justified by the fact that the pressing-in does not work for a set of faces composing a cycle.

The correctness of the algorithm can be shown as follows. The first step to create a polyhedral surface $\hat{H}$ which separates the voxels of a given object S and its complement $A \setminus S$ is to compute the convex hull $CH(S)$. The correctness of the convex hull algorithm has already been shown in [8]. When the convex hull is computed two cases can occur: (1) $CH(S)$ does not contain any voxel of the set $A \setminus S$ and thus it is convex and (2) $CH(S)$ contains voxels of $A \setminus S$. In the first case the convex hull is a polyhedron which separates the voxels since no voxel of S lies outside $CH(S)$ and no voxel of $A \setminus S$ lies inside, and thus the algorithm stops here. In the second case the algorithm detects the class of the background voxel components and if these components are concavities it modifies the convex hull using the concept of pressing-in. This procedure moves all voxels of a concavity S' to the outside of the current polyhedron P through a substitution of faces of P with a new set of faces which lies completely inside P. This is done recursively for each concavity. In each step the algorithm deals with a current polyhedron P and a second smaller polyhedron P' lying completely inside P. The second polyhedron is smaller than the first one in the sense that it includes a fewer number of voxels. This means that by reducing the number of voxels in the current polyhedron the number of recursive steps is finite and the algorithm stops after a finite number of steps with a separating polyhedron $\hat{P}$ as its result.

4.3. Degree of non-convexity

In Section 4.1 we have already presented a classification of non-convex objects regarding the number of connected sets of faces of the convex hull into concavities, cavities and tunnels. The number of such components of background voxels inside the convex hull can be used as a measure to express the complexity of the object. But solely using this measure does not distinguish between components of different complexity. For a good classification of the complexity of the objects we need a second measure which expresses the intricacy of the background voxel components.

For this purpose we introduce the degree of non-convexity, defined as follows:

Definition NK. The *degree of non-convexity* $G_{nc}(S)$ of a non-convex subcomplex S of an AC complex A is defined recursively:

(1) For every convex object S it holds:

$$G_{nc}(S) = 0. \tag{1}$$

(2) The degree of non-convexity of an object S which is non-convex and contains exactly one component S' of background voxels, is greater than the degree of non-convexity of S' by one:

$$G_{nc}(S) = G_{nc}(S') + 1. \tag{2}$$

(3) The degree of non-convexity of an object S which is non-convex and contains more than one component of background voxels, is greater than the maximum degree of non-convexity of its background voxel components by one:

$$G_{nc}(S) = \max_i\{G_{nc}(S'_i)\} + 1. \tag{3}$$

Since the degree of non-convexity is defined without being restricted to concavities, it can be applied to all three classes of background voxel components.

5. Complexity

5.1. Complexity of the convex hull algorithm

O'Rourke has shown that the complexity of an incremental convex hull algorithm in three dimensions is $O(v^2)$, where v denotes the number of vertices of the polyhedron [10]. Our method is also an incremental algorithm and all vertices of the convex hull are voxels of S and therefore the complexity of the algorithm can be considered as $O(n^2)$, where n denotes the number of voxels of S.

Intuitively one could presume that the algorithm has linear time complexity since it has to run through the list L of voxels only once when spanning the convex hull. But during the steps of the extension procedure the number of faces to be tested increases linearly and leads to the overall time complexity of $O(n^2)$.

5.2. Complexity of the modification algorithm

Let n be the number of voxels of the object S, k the number of concavities inside $CH(S)$ and $m = G_{nc}(S)$ the degree of non-convexity of S (see Section 4.3). The complexity of the modification algorithm is composed of the following parts: Inside the `modify_polyhedron` subroutine (see Algorithm 1) we have to find the concavities, which can be done going twice through the list of all voxels, i.e. in $O(n)$ time. For each of the k concavities we have to create the convex hull. As already mentioned, this is of order $O(n^2)$. Merging polyhedra as well as changing roles of object and background voxels has a complexity of $O(n)$. Since `modify_polyhedron` is called recursively the complexity of the whole routine is $O(m \cdot k \cdot n^2)$, but in practice it can be observed that it behaves as one of quadratic complexity since m as well as k is much less than n.

Illustrative examples can be found in [12].

6. Conclusion

In this paper we present a new algorithm for computing polyhedral surfaces approximating non-convex three-dimensional digital objects represented as a strongly connected set of voxels. The resulting polyhedral surface is an abstract polyhedron which is a particular case of an abstract cell complex. The polyhedron is encoded by the non-redundant version of the two-dimensional cell list which is a good tool to save topological and geometric information efficiently and without redundancy. The cell list also provides the possibility of exactly reconstructing the voxel object.

Furthermore we present some improvements to our convex hull algorithm which lead to faster computation through sorting the list of vertex candidates. We also discuss the three variants to treat coplanar faces and we show that the extension method is the fastest one.

The algorithm presented in this paper still needs further development. There are some drawbacks, especially the one mentioned in Section 4.2, concerning surfaces with a genus greater than 0. Also the labeling criteria are still under investigation.

The algorithm can be applied in a variety of tasks. Especially in the field of three-dimensional image analysis and computer graphics it can be used to visualize voxel sets by polyhedra and to store large sets of voxels efficiently and without any loss of information.

References

[1] P.K. Agarwal, S. Suri, Surface approximation and geometric partitions, SIAM Journal on Computing 27 (4) (1998) 1016–1035.
[2] H. Brönnimann, M.T. Goodrich, Almost optimal set covers in finite VC-dimension, Discrete and Computational Geometry 14 (1995) 263–279.
[3] P. Cignoni, C. Montani, R. Scopigno, DeWall: A fast divide & conquer delaunay triangulation algorithm in E^d, Computer Aided Design 30 (5) (1998) 333–341.

[4] G. Das, M.T. Goodrich, On the complexity of optimization problems for 3-dimensional convex polyhedra and decision trees, Computational Geometry: Theory and Applications 8 (1997) 123–137.
[5] V.A. Kovalevsky, Finite topology as applied to image analysis, Computer Vision, Graphics and Image Processing 45 (2) (1989) 141–161.
[6] V.A. Kovalevsky, A topological method of surface representation, in: G. Bertrand, M. Couprie, L. Perroton (Eds.), Discrete Geometry for Computer Imagery, in: Lecture Notes in Computer Science, vol. 1568, Springer-Verlag, 1999, pp. 118–135.
[7] V.A. Kovalevsky, Algorithms in digital geometry based on cellular topology, in: R. Klette, J. Žunić (Eds.), Combinatorial Image Analysis, in: Lecture Notes in Computer Science, vol. 3322, Springer-Verlag, 2004, pp. 366–393.
[8] V.A. Kovalevsky, H. Schulz, Convex hulls in a 3-dimensional space, in: R. Klette, J. Žunić (Eds.), Combinatorial Image Analysis, in: Lecture Notes in Computer Science, vol. 3322, Springer-Verlag, 2004, pp. 176–196.
[9] W.E. Lorensen, H.E. Cline, Marching cubes: A high-resolution 3D surface construction algorithm, Computer Graphics 21 (4) (1987) 163–169.
[10] J. O'Rourke, Computational Geometry in C, Cambridge University Press, 1994.
[11] H. Schulz, Polyhedral surface approximation of non-convex voxel sets through the modification of convex hulls, in: V.E. Brimkov, R.P. Barneva, H.A. Hauptman (Eds.), Combinatorial Image Analysis, in: Lecture Notes in Computer Science, vol. 4958, Springer-Verlag, 2008, pp. 38–50.
[12] H. Schulz, Polyhedral surface approximation of non-convex voxel sets and improvements to the convex hull computing method, Wissenschaftlich-Technische Berichte, Forschungszentrum Dresden-Rossendorf, FZD-514, 2009. http://www.fzd.de/publications/012497/12497.pdf.
[13] S. Svensson, C. Arcelli, G. Sanniti di Baja, Characterising 3D objects by shape and topology, in: I. Nyström, G. Sanniti di Baja, S. Svensson (Eds.), Discrete Geometry for Computer Imagery, in: Lecture Notes in Computer Science, vol. 2886, Springer-Verlag, 2003, pp. 124–133.